Quick and Automatic Selection of POMDP Implementations on Mobile Platform Based on Battery Consumption Estimation

by

Xiao Yang

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2014

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Partially Observable Markov Decision Process (POMDP) is widely used to model sequential decision making process under uncertainty and incomplete knowledge of the environment. It requires strong computation capability and is thus usually deployed on powerful machine. However, as mobile platforms become more advanced and more popular, the potential has been studied to combine POMDP and mobile in order to provide a broader range of services. And yet a question comes with this trend: how should we implement POMDP on mobile platform so that we can take advantages of mobile features while at the same time avoid being restricted by mobile limitations, such as short battery life, weak CPU, unstable networking connection, and other limited resources.

In response to the above question, we first point out that the cases vary by problem nature, accuracy requirements and mobile device models. Rather than pure mathematical analysis, our approach is to run experiments on a mobile device and concentrate on a more specific question: which POMDP implementation is the "best" for a particular problem on a particular kind of device. Second, we propose and justify a POMDP implementation criterion mainly based on battery consumption that quantifies "goodness" of POMDP implementations in terms of mobile battery depletion rate. Then, we present a mobile battery consumption model that translates CPU and WIFI usage into part of the battery depletion rate in order to greatly accelerate the experiment process. With our mobile battery consumption model, we combine a set of simple benchmark experiments with CPU and WIFI usage data from each POMDP implementation candidate to generate estimated battery depletion rates, as opposed to conducting hours of real battery experiments for each implementation individually. The final result is a ranking of POMDP implementations based on their estimated battery depletion rates. It serves as a guidance for on POMDP implementation selection for mobile developers.

We develop a mobile software toolkit to automate the above process. Given basic

POMDP problem specifications, a set of POMDP implementation candidates and a simple press on the "start" button, the toolkit automatically performs benchmark experiments on the target device on which it is installed, and records CPU and WIFI statistics for each POMDP implementation candidate. It then feeds the data to its embedded mobile battery consumption model and produces an estimated battery depletion rate for each candidate. Finally, the toolkit visualizes the ranking of POMDP implementations for mobile developers' reference.

Evaluation is assessed through comparsion between the ranking from estimated battery depletion rate and that from real experimental battery depletion rate. We observe the same ranking out of both, which is also our expectation. What's more, the similarity between estimated battery depletion rate and experimental battery depletion rate measured by cosine-similarity is almost 0.999 where 1 indicates they are exactly the same.

Acknowledgments

I would like to thank all the people who made this possible, including but not limited to Jesse Hoey, Pascal Poupart, and Marek Grzes.

Dedication

This is dedicated to the one I loved.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Partially Observable Markov Decision Process (POMDP)[28] is a powerful and widely used mathematical model for sequential decision making. It is capable of capturing stochasticity and uncertainty in real world situations, and selects probabilistically optimal actions based on a history of actions and observations of the external environment.

Generally, the richer the structure we allow POMDP to have, the more competent it will be when interacting with the real world. Yet the richer its structure grows, the more complicated its computation becomes. That's why POMDP is normally run on work stations or PCs. Because they have powerful computation resources to explore the advantages of structurally enriched and thus computationally intensive POMDP. Though deployment on large machines solves the computational need, it restricts POMDP's portability, adaptability, and consequently its availability. In some circumstances, we need POMDP to "move around" handling context-aware signal such as GPS data, accelerometer measurement, voice input, etc.[18]. POMDP tied to cumbersome machines are not directly available for these applications.

Fortunately, the development of mobile technology opens another door for us. With

an emerging generation of mobile platforms capable of resolving some of the heavy computation, the deployment of POMDP on portable devices is not out of reach. If we can successfully embed POMDP in mobile platform, not only will POMDP utilize the flexibility and ubiquity of mobile devices to largely extend their application scope, but also applications on mobile platforms can benefit from POMDP to provide more sophisticated and considerate services.

In order to deploy POMDP on mobile, we need to have basic understanding of how POMDP works. The general working logic of POMDP is two-phase. A policy needs to be computed at first and the actual execution is essentially state updating and action querying according to the policy. Usually, the POMDP policy computation is a much heavier job that is not suitable for mobile devices. A commonly adopted approach[6] is to compute the policy on PCs and download it to mobile devices. After all, the mobile features are useful only during policy execution. In such case, the mobile platform only serves as a carrier of POMDP's execution, and acts according to a pre-calculated policy. Though in some circumstances, we may even want to compute policies using a mobile device, but we don't address it in this thesis. A promising future is lying ahead; however, the reality is that mobile platforms are still far from being as powerful as PCs. Sometimes, POMDP's execution itself is already hard even for PCs, let alone their comparably delicate counterparts. Therefore, further considerations are necessary to optimize POMDP's execution on mobile devices. POMDP has variable implementation methods. Since POMDP's execution has to happen on mobile devices, one unavoidable question is how to trade off among different POMDP implementations. And it turns out to matter a lot for POMDP's performance on mobile devices, which we will discuss later.

How do we select proper POMDP implementations? For the above question, a lazy answer is that it depends. Indeed, the selection of POMDP implementations depends on various factors, such as problem nature, policy complexity, mobile device model, and so on. We admit that attempts to formalize this question in pure mathematics would mostly fail

or yield an imprecise result. But there is certain thing we can do to rationalize our selection from another perspective. That is through benchmarking, which means we conduct experiments for each POMDP implementation candidate and select the one with the "best" experimental performance. Though pre-test is not as simple as mathematical formulas, it is definitely the most straightforward and realistic reflection of how POMDP implementation would perform on mobile devices. Benchmarking is a little time-consuming, but the time it costs is worthwhile since once a POMDP is embedded in a mobile device, it is expected to run for a considerable length of time. Thus, it is absolutely intelligent to spend some time on a benchmarking and choose a better implementation for the future well-being in the long run.

We select POMDP implementation for the good of mobile user experiences. Users don't care how POMDP is implemented. Assuming our implementation candidates can provide the same level of accuracy in reasonable a response time (those implementation that can not achieve this should be ruled out at the first place during our implementation candidates selection), all users care about is how running this POMDP application would affect their mobile devices. The most visible effect is the battery consumption. According to smartphone survey[8], battery consumption is one of the top issues that concerns mobile users. The other top ones (such as signal strength, crashing down, etc.) have nothing to do with which particular POMDP implementation we choose. And some survey[25] even reveals that short battery life is the top one gripe in mobile users experience. Therefore, our POMDP implementation selection is mainly based on the battery consumption on mobile devices. We also ignore those factors which appear to be related to user mobile experiences but in fact have no influence on our POMDP implementation selection, such as the usage of different sensors, user interface. It is because for a particular application, no matter which POMDP implementation we prefer, we always have to provide the same input set and visualize the output in the same way. In conclusion, the POMDP implementation is transparent to mobile users and the most visible impact is the battery draining, which

is also what we focus on to minimize. It also makes it clear for the research purpose because we now have more focused problem - selection based on battery consumption. We use average battery depletion rate to represent the battery consumption performance. Obviously, the lower the battery depletion rate, the better this POMDP implementation is. In this case, the "goodness" of POMDP implementation is quantified by a single number. This self-contained criterion makes it easier for us to rank different implementations.

A naive and inefficient way to obtain mobile battery depletion rate is to run real battery experiments for each individual POMDP policy execution implementation. We propose a more intelligent method - a mobile battery consumption model. It translates CPU usage, WIFI connection into part of the battery depletion rate. According to mobile battery research[5], CPU and WIFI usages are the main factors that affect mobile battery consumption (other main factors such as signal standby, screen brightness are not related to POMDP implementation selection and thus ignored). With this mobile battery consumption model, we only need one real battery test for the device condition (we define it as the benchmark), and for each POMDP implementation candidate, we simply record CPU and WIFI status, which can be done in seconds and save us from hours of tedious battery experiments. The benchmark and CPU, WIFI data are combined within the mobile battery consumption model to produce an estimated battery depletion rate for each POMDP implementation as their overall battery performance on mobile devices.

We develop a software toolkit that automates the above process and makes a reasonable and quick decision among multiple POMDP implementations. In order to obtain some basic information of target platform, the toolkit conducts a set of benchmark experiments, followed by small and quick CPU and WIFI measurement for each implementation candidate. Then it feeds the results to our mobile battery consumption theory model to generate the estimated battery depletion rate one for each POMDP implementation. At last, it produces a ranking as selection suggestion, with auxiliary information, including memory usage, WIFI necessity. The toolkit provides a comprehensive suggestion about

POMDP implementation selection on mobile devices. It automatically finishes all required steps. All users need is to install our toolkit on a target mobile device, provide necessary POMDP specifications, launch the toolkit, and wait for results.

Evaluation is done by comparing our estimated battery depletion rate with real battery depletion rate measured by running real POMDP applications and recording battery changes. Our expectation is to see the ranking of POMDP implementations derived from these two battery depletion rates are similar. It turned out better than we have expected. Our estimated battery depletion rates well predict the real depletion rates. Not only is the ranking based on our estimation the same as that from real battery experiments, but also the difference between these two figure set - estimated battery depletion rate and real battery depletion rate - is almost 0. They have around 0.999 cosine-similarity where 1 indicates that they are exactly the same.

The primary contributions of this thesis are six-fold:

1. We point out the importance of selecting POMDP implementations for mobile platform and suggest the selection should be based on experiments.

2. We design and justify criteria of selecting POMDP implementations for mobile based mainly on battery consumption.

3. We propose a method to quickly estimate battery consumption in the long run for POMDP on mobile as opposed to making time-consuming measurements.

4. We build a framework to conduct battery experiments on mobile devices for POMDP problems and show that it can be generalized to other similar applications.

5. We implement a toolkit that automates the selection of POMDP implementations.

6. We explore the feasibility of loading computational heavy tasks onto mobile platforms.

The remainder of the thesis is organized as follows: Chapter 2 refreshes our memory of POMDP knowledge and indicates advantages using Android platform; Chapter 3 compares our work with related research, and identifies connections and our distinctiveness; Chapter 4 provides a comprehensive demonstration of our overall design of the toolkit, including selection criterion, mobile battery consumption model, and system architecture; Chapter 5 presents a clear framework along with implementation of our toolkit; Chapter 6 illustrates how evaluation is conducted and shows the results are as expected; Chapter 7 concludes the whole thesis and points out future directions.

Chapter 2

Background

2.1 MDP Recap

Speaking of the POMDP, we first need to refresh our memory of Markov Decision Process (MDP)[30]. MDP is a mathematical framework use to capture sequential decision process under uncertainty. It models how an agent interact with the outside world. The agent's status is described through the concept of "state". At a state, the agent can perform different actions and go to the potential next state. The uncertainty here is because after performing an action, the agent is not guaranteed to land on a certain next state. Instead, there is a distribution of the next state given the current state and performed action. There is a reward system which quantifies the goodness of the agent's single decision, based on the current state and chosen action. The agent's goal is to maximize the long term rewards during either a fixed or infinite number of actions to perform. Mathematically, the MDP consists of the following:

- A finite set of states: S;
- A finite set of actions: A;

- A state transition distribution: $T(s^{t+1}, s^t, a^t) = Pr(s^{t+1}|s^t, a^t)$ $(s^{t+1}, s^t \in S, a^t \in A)$, probability of landing on s^{t+1} given the current state s^t and chosen action a^t;
- A reward system: $R(s^{t+1}, s^t, a^t)$, reward for performing action on current state s^t and landing on next state s^{t+1}

2.2 POMDP Framework

Partially Observable Markov Decision Process (POMDP)[28] is a more generalized form of MDP. POMDP extends Markov Decision Process (MDP) to integrate another level of uncertainty. In MDP, only the transactions between states resulted from performing actions are probabilistic. The external environment is fully-observable. In POMDP, even the external environment is not fully observable and thus also uncertain. This is often the case in real world while an agent is making a decision. The agent doesn't know for sure which state it is in, and can only infer from what it observes. For example, in a simple navigation problem, an agent in a room wants to reach another location in this room. But its vision is blocked by furniture that it doesn't know exactly where in the room it currently stands. However, it can make a guess based on the surrounding environment (e.g. chair, desk on its left) to come up with a probability distribution over some potential possible locations. Since that is all it has, it should make decision based on the likelihood of its current state. By introducing the concept of observation as a belief of external environment, POMDP provides a more general method to model real world sequential decision making. Mathematically, the POMDP consists of the following:

- A finite set of states: S;
- A finite set of actions: A;
- A finite set of observation: O

- A state transition distribution: $T(s^{t+1}, s^t, a^t) = Pr(s^{t+1}|s^t, a^t)$ $(s^{t+1}, s^t \in S, a^t \in A)$, probability of landing on s^{t+1} given the current state s^t and chosen action a^t;
- A observation distribution: $Z(o^t, s^t, a^{t-1}) = Pr(o^t|s^t, a^{t-1})$ $(s^t \in S, a^{t-1} \in A, o^t \in O)$, probability of obtaining observation o^t given current state s^t and previous action a^{t-1};
- A reward system: $R(s^{t+1}, s^t, a^t)$, reward for performing action on current state s^t and landing on next state s^{t+1}

2.3 Policy Representation

An agent's goal in sequential decision, either MDP or POMDP, is to select actions that maximize long term rewards (including the immediate reward of executing an action and the potential future reward because of the direction this action points to). The long term means during the next t steps (finite horizon) or all of the future steps (infinite horizon). During this long term period, we would need a strategy that help we choose action. This is called policy, and denoted by π. In MDP, because of the Markov property, namely the current situation only depends on the previous stage, the policy usually is a simple mapping from states to actions: $\pi : s \rightarrow a$. The long term reward can be expressed in a value function that associates with state and policy. Starting from state s^t and with policy π, the long term reward is:

$$V(s^t, \pi(s^t)) = R(s^t, \pi(s^t)) + \gamma \sum_s (T(s^{t+1}, s^t, \pi(s^t)) * V(s^{t+1})) \tag{2.1}$$

The optimal policy is the one achieve maximum V. γ here is a discount factor that characterizes the preference between earlier reward and future reward ($0 \leq \gamma \leq 1$). It also mathematically converges the value function.

Such straightforward mechanism is directly applicable to MDP but not for POMDP. In POMDP, the agent doesn't have complete knowledge of the current state. Instead, it can only observe what's going on "around" it. One time observation is too vague to determine the real world situation. Therefore, in order to make good decision, the agent has to consider what observations it has obtained and actions it has performed previously, that is the history of observations and actions. It is intuitively straightforward but not mathematically easy to solve because the Markov property no longer holds. The policy for POMDP has to map from history to actions. It is conceptually a tree structure as shown in Figure 2.1.

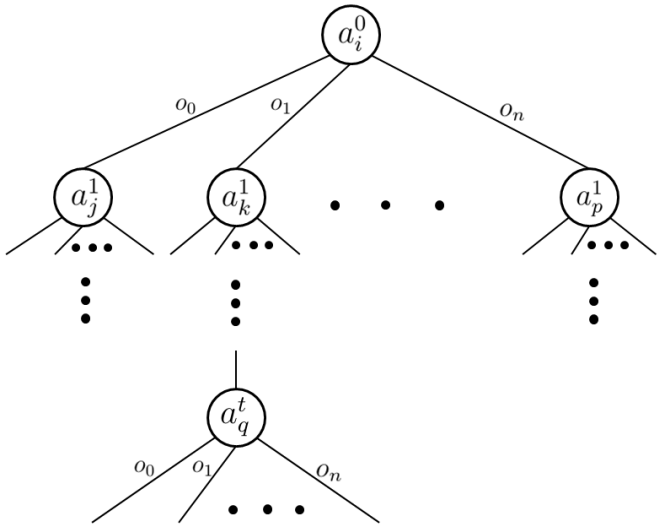

Figure 2.1: POMDP Policy Tree

Each node is labeled with the action to perform. The agent starts from an initial state or initial action, goes along with the obtained observation and performs the actions on the visited nodes. If the agent is only looking for the next t step, the depth of the tree is t, if it is infinite horizon, the policy tree is infinite large. But even with finite t steps, the

tree is also exponential to observations. And there are $|A|$ potential actions to choose at each node, which further complicates the structure of the policy tree. Therefore, we need a smarter way to represent POMDP policy.

2.3.1 α-vector

A straightforward way is to transform the state to a form that the Markov property still holds. It turns out that we can apply a belief state which is a distribution over all states. Its dimension is the size of state set. On each dimension $b(s)$ indicates the probability the agent is in state s. Of course $\sum b(s) = 1$. The belief state update rule is:

$$\begin{aligned} b^{t+1}(s') &= \frac{Pr(o|s',a)\sum_s Pr(s'|s,a)b^t(s)}{Pr(o|b^t,a)} \\ &= \frac{Z(o,s',a)\sum_s T(s',s,a)b^t(s)}{Pr(o|b^t,a)} \end{aligned} \tag{2.2}$$

$$\begin{aligned} Pr(o|b^t,a) &= \sum_s Pr(o|s',a)\sum_s Pr(s'|s,a)b^t(s) \\ &= \sum_s Z(o,s',a)\sum_s T(s',s,a)b^t(s) \end{aligned} \tag{2.3}$$

Belief state is equivalent to maintaining a whole history of observations and actions[35]. Now, we have transformed a partially observable MDP to a completely observable MDP where the Markov property holds. Thus, our value function is:

$$V(b^t) = \sum_s b^t(s)R(s,a) + \gamma \sum_s Pr(o|b^t,a)V(b^{t+1}) \tag{2.4}$$

Apparently, the policy achieves maximum value is our optimal policy. The next thing is to find a mapping from belief state to action. Now that the state is continuous, the simple mapping table we use for MDP no longer works. However, the value function over belief

states is piece-wise linear (finite horizon) or close to piece-wise linear (infinite horizon)[35]. That means we can use a set of vectors, called α-vectors[10], to express the value function:

$$V(b) = \sum_i \alpha_i V(b_i) \tag{2.5}$$

Each vector is associated with a policy tree (finite horizon) or an action(infinite horizon[35]). For finite horizon, the agent starts from b^0, and finds the optimal policy tree, then executes action contingent on observations it receives. For infinite horizon, the agent updates the belief state based on previous belief state, action performed and observation received. Then it chooses the next action that yields the best value. In either case, the essential step is the same and simple: find the vector that yields the maximum value based on current belief state. The entire α-vector set is our policy representation.

2.3.2 Finite State Controller

Sometimes the policy tree in Figure 2.1 doesn't have to keep growing[20]. If the finite horizon value function $V^t = V^{t+1}$, we can redraw the policy tree so that from one level, nodes can point to itself or previous nodes. Then we can convert non-stationary policy tress into stationary cyclic policy graphs, which enables an agent to execute policies simply by doing actions prescribed at the nodes, and following observation links to successor nodes [2]. Therefore, there is no need for that updating belief state and computation burden of selecting optimal vectors. We can maintain an internal state set which is completely observable. Things all go back to nice completely observable MDP as shown in Figure 2.2.

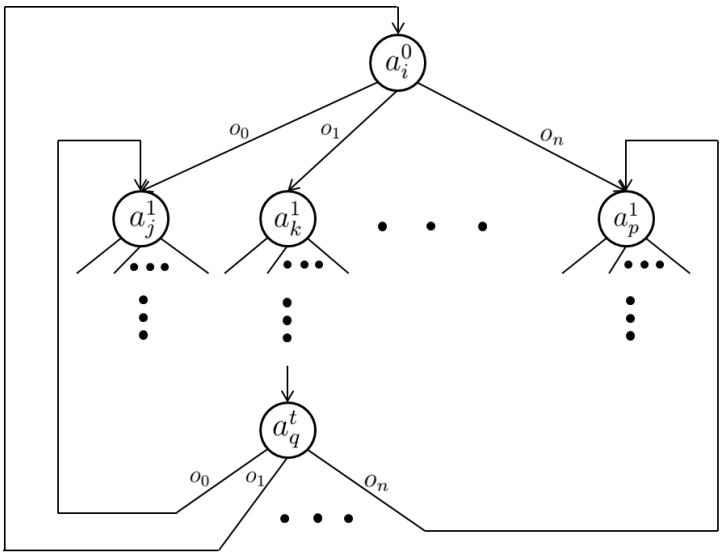

Figure 2.2: Example of Redrawing Policy Tree From Figure 2.1 to A Completely Observable Policy Graph

2.4 POMDP Specification Grammar

We need a practical way to specify POMDP so that it can be understood by machine. The nature of POMDP can be expressed in a hierarchy structure as shown in Figure 2.3. We have four tree structures, each of them to represent states S, actions A, observations O, and reward R. Then the tree structure can be serialized into parentheses statements, using parentheses to recursively indicate hierarchy, e.g. $treeRoot(subTree1, subTree2, ...)$. What's more, there is a technique called Stochastic Planning using Decision Diagrams (SPUDD) that can calculate policies based on the hierarchy structure[16].

1. The *state tree* consists a possible states. The root of the *state tree* has n branches, each of which represent the variable in one dimension of the state. The variable in

each dimension may have several discrete values to choose from. Therefore, the state set can be represented in plain text such as

$State(Variable_1(Value_1, Value_2, ...,), ..., Variable_n(...))$.

2. The *action tree* not only contains all the actions, but also encodes the state transition probability. By going from one action in the second level to one particular leaf node, we will obtain the transition probability of performing that action in a current state and arriving at the a next state. The whole action tree is essentially tree-structural coding of the transition probability. The action set is represented in plain text as $Action(a_1(s_1(s_2(T), ...)), ...)$.

3. The *observation tree* is similar to the action tree. Observation distribution is encoded along the branches to leaves. The observation set is represented in plain text as $Observation(o_1(s_1(s_2(Z)...)), ...)$.

4. The *reward tree* is also similar to the action tree. And it can be represented in plain text as $Reward(a_1(s_1(s_2(R)...)), ...)$.

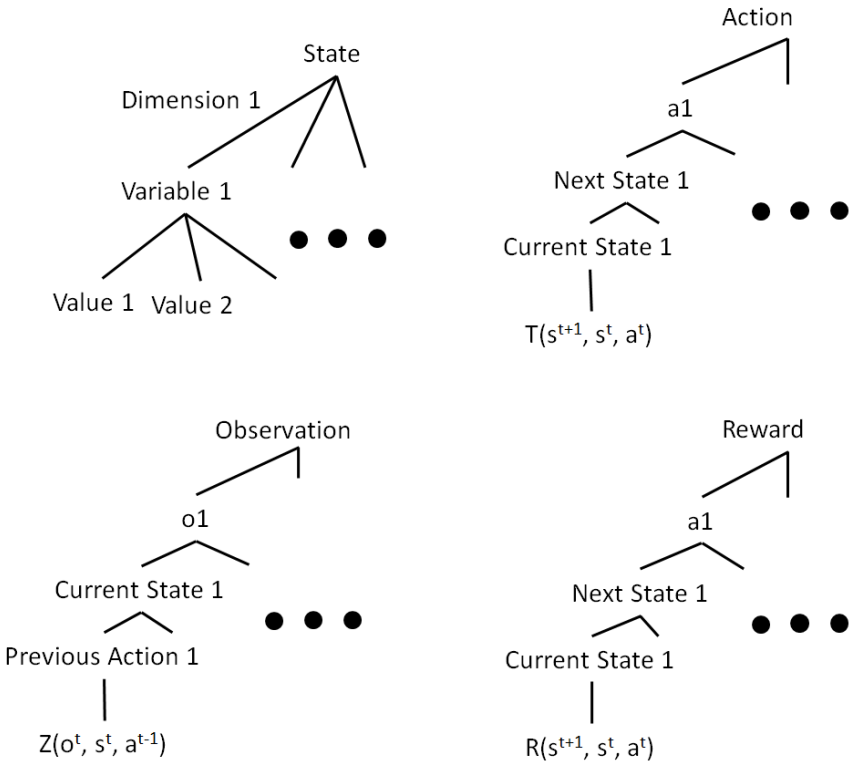

Figure 2.3: POMDP specifications

2.5 Android Platform

We choose Android platform[4] as our primary interested target for implementation and experiments for the following reasons:

- Android is widely popular for mobile device. Not just mobile phone and tablet, but also includes televisions, games consoles, digital cameras and other electronics. Based on it, we can easily extend our service to a broad range of devices.
- Android has 80% of market share and it's still growing very fast. A large community

is studying and developing apps on Android. And we would like to have our toolkit known by more people including potential users and developers.

- Android is an open-source project suitable for research purposes.
- The primary developing languages on Android is Java, which allows us to plug in plenty available POMDP packages, e.g., libpomdp[1], Symbolic Perseus[2], RL-POMDP[3], etc..

[1]Diego Maniloff, libpomdp, 2010, https://github.com/dmaniloff/libpomdp
[2]Pascal Poupart, Symbolic Perseus, 2009, https://cs.uwaterloo.ca/ ppoupart/software.html
[3]Qiming He, RL-POMDP, Version 1.0, 2013, https://sourceforge.net/projects/rl-pomdp/

Chapter 3

Literature Review

Partially Observable Markov Decision Process (POMDP) has been used in many real life applications. For example, White[39] applied POMDP to questionnaire design. The application decides the sequence of questions to ask based on how the participants respond in hope of getting more genuine answers. Crites[9] deployed POMDP onto elevator control system. Given the up and down buttons pressed or not, it makes decision about which elevator to send and where it stops by trying to model passengers' desired destinations. Bandera[1] employed POMDP to understand images with some low-resolution areas so that resource demand can be alleviated because in this case, images don't have to be recorded with uniform high-resolution. Hauskrecht[15] used it to assist medical diagnosis. It tries to minimize the cost incurred by performing diagnosing actions through studying observed symptoms. Williams[40] applied POMDP to spoken dialogue system to unify and extend the existing framework, which yielded significant quantitative gains in empirical results. The COACH (Cognitive Orthosis for Assisting with aCtivites in the Home) project[38] built a prototype of an intelligent assistant for people with dementia at home with POMDP model running as its core. This assistant will track their behaviors, plan for a better solution and prompt when necessary. One concrete example of the COACH

project is Hoey's handwashing system[17]. It installed a camera to track users' handwashing progress. When unexpected patterns detected, the system would analyze this exception and using voice prompts to remind users what might be the right things to do.

The above applications are implemented designated for computationally powerful machines. With the recently boost of mobile technology, researchers start to realize the possibility as well as advantages of combining PODMP with mobile devices. Hoey's LaCasa[18] is good example of such attempt. LaCasa is designed to help patients with a cognitive disability to find their way home. It gathers users' location, proximity to the caregivers, surrounding noise, etc. to figure out a considerate way to help users. POMDP is the core of the application. It is deployed on mobile devices carried by users to analyze real-time information, perform opportune modeling of users' status and provide timely assistance. Another example of POMDP on mobile is Pollack's Pearl[26]. They developed a nursebot on mobile to remind people about routine activities such as eating, drinking, taking medicine, and use the bathroom. The modeling tool is also POMDP. And it also need to be "around" users all the time so that it can monitor users' status.

Mobile platform offers a portable way to carry POMDP and extend its availability to a broader range of potentials. However, such potentials come with limitations. Compared to large machines, mobile devices are unstable, of less powerful CPU and limited memory, and known for short battery life, among which the short battery life concerns us the most. According to smartphone usage experiences[8], short battery life ranks in the top class of most disturbing issues during mobile usage experiences. And it is the problem we focus on in this thesis because other top ones such as lengthy loading times, no access, crashing and slowing down are often very rare but short battery life is too appreciable to be ignored. It is especially for application like POMDP which consumes lots of computational resources.

Researchers have long realized the battery life problem. In fact, it doesn't just apply to POMDP. Any mobile application demanding large resources would suffer from the same

dilemma - trying to utilize mobile features while trying to avoid the consequent battery life restriction. Some researches have been conducted to study and model mobile battery behaviors. Raghunathan[31] addressed the modeling of mobile battery from the perspective of hardware. They analyzed from the circuit level to derive methods that can simulate voltage change. Carroll's work[5] consisted of real battery experiments. It performed a comprehensive set of battery tests on multiple mobile component, including CPU, WIFI, signal standby, and screen to reveal proportion of battery consumption on each component. It is very useful information for understanding where the mobile energy is consumed. Flinn[11] completed mobile battery modeling by not just observing it, but also presenting a well-designed and implemented profiling method to map energy consumption to program structure.

Given these studies, some solutions are found to ease the battery issue. Three typical types of solutions are popular:

1. One brute force approach is to transfer all computation to the cloud. CloneCloud[7] presented a method to dynamically identify computing-intensive blocks from the Java Virtual Machine level and shift these blocks to the cloud. In this case, the mobile device serves more like a virtual screen for visualizing what's happening remotely in the cloud. It is quite an efficient solution regarding alleviating mobile computation burden but not necessarily the optimal choice for saving battery. Because the network communication is also energy-consuming, especially when transmitting large amount of data for a long time. A improved method which we examined in previous work is taking network communication cost as considerations and decide when to shift what to the cloud (AppendixB). It abstracted program's execution as sequence of blocks. Each block takes input, does computation, and produces output. Blocks are originally on mobile but can be transferred to the cloud. By considering computation and network transferring energy consumption, it assigns and tags blocks as "for mobile"

or "for the cloud" before programs' executions and executes it accordingly.

2. Some concentrate on mobile platform level to improve battery life. Prabhu[29] did that in pure theoretical way. In their work, the mobile battery is treated as a server with finite service capacity. They proposed a queuing theory along with complicated formula to exploit a recharge phenomenon[21, 24]. They discovered that allowing intentional vacations during busy periods helps increase battery life. Rulnick[33] introduced a system level power management idea. They derived and tested an algorithm addressing how mobile devices can adjust their transmitter power in wireless network to lower battery consumption. Their result indicated the possibility of substantially reducing energy consumption without sacrificing quality of service, and possibly enhancing network stability and capacity along the way.

3. Others focus on the application design side. Edward[3] proposed an innovative approach to understand and quantify average current battery drain. Their battery drain analysis can provide insight to optimize data transmitting and processing, as well as operating time for application design. Sharkey[34] listed rules mobile developers should follow on the coding level based on statistics and experiences. Rules suggest checking network connection before using, avoiding wakelock, creating less garbage during Java developing, shortening background service life time, and many others. They can all greatly save mobile energy consumption. Narayanan[23] provided a history-based mechanism to predict CPU usage. The prediction can be used to improve mobile applications' adaptation to mobile resources, including battery. Flinn[11] studied smartphone user behaviors and concluded some user pattern that developers can take advantages of during mobile application design and implementation to save more battery consumption.

The above researches are interesting and helpful. But to the best of our knowledge, our work has some advantages that others don't have. We start from a different angle -

instead of making general assessment of mobile battery, we specifically address POMDP battery issue and gradually build a framework that can be generalized to apply to other computationally intensive applications. We point out that even different implementations can cause significant battery life differences and it is worthwhile to spend time making a proper selection. We don't mind measuring from the application layer which is often considered imprecise. We think it is in fact the most realistic reflection of mobile battery performance. We implement a software toolkit to automate POMDP implementation selection process and make the best choice in terms of saving mobile energy.

Chapter 4

Design

4.1 POMDP on Mobile

The most obvious approach to combine POMDP and mobile devices would be client/server mode, in which mobile devices just act like clients serving the purpose of collecting environmental situation and communicating with POMDP servers. The POMDP servers handle most of the computation, including calculating the policy, reacting to action query and updating states. This is because even though the development of mobile technology is inspiring, it is not yet powerful enough to bear the whole POMDP computation. With the client/server mode, we can have a virtual POMDP on mobile devices to utilize those mobile features, such as portability and rich environmental sensors. However, this solution only works when network connection is guaranteed, which is not always true because exceptions happen a lot on mobile devices and sometimes, there is even no network connection at all.

The above situation leads to another way of deploying POMDP on mobile devices. Since it is difficult and of no need to compute POMDP policies on mobile devices, we can compute POMDP policy elsewhere and download the policy onto mobile devices. Rather

than totally relying on mobile devices for both policy computation and execution, we will use mobile devices only as carriers for the POMDP policy execution. This approach does have some shortcomings, such as extra efforts of downloading POMDP policy specification files and lack of flexibility when we want to modify the POMDP model and consequently needs modification of the downloaded POMDP policy. Nevertheless, it relaxes the network requirement and creates a stand-alone and self-contained POMDP entity on mobile devices, which can be generalized to other devices such as embedded chips.

Now, there has already been choices between client/server mode and mobile-only mode. We are more ambitious than that. Since even for the mobile-only mode, there are plenty of different implementation methods worth studying. We intend to examine various implementations and identify the best one according to different scenarios. To simplify the problem, we assume that in client/server mode, every POMDP implementation method has the same effect on mobile client. That is to say, from the perspective of mobile devices, the POMDP implementation on server is transparent to them, and thus no need to take it into consideration.

4.2 Abstraction of POMDP Execution

After policy computation is performed somewhere else, the essential part is the execution. POMDP execution seems to be complex. But in fact, in a high level, it can be abstracted into a concise step by step model, as shown in Figure 4.1.

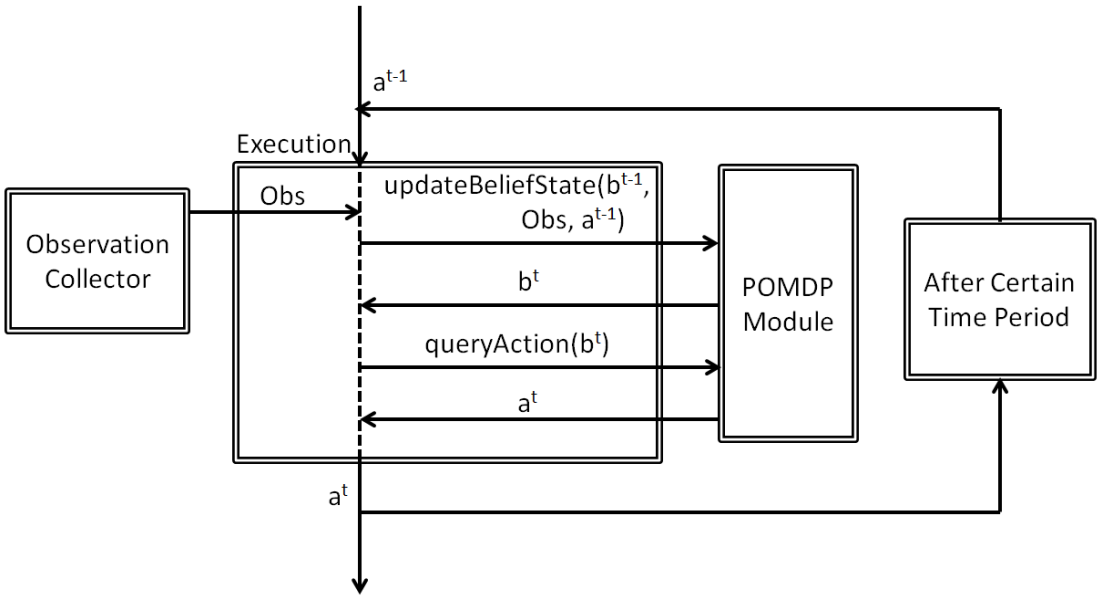

Figure 4.1: POMDP Execution Abstraction

Generally, POMDP execution has to: first, obtain observation of current situation; second, update current belief state based on previous action and current observation; third, obtain action that maximizes overall rewards given the belief state. Since POMDP is a sequential decision model, after certain time period, the execution repeats the above simple three-step process. It will run in a infinite loop or to some pre-defined horizon. During each round, different implementations vary in the POMDP module in the above figure. POMDP module may locally or remotely, provide *udpateBeliefState*(b^{t-1}, Obs, a^{t-1}) and *queryAction*(b^t) interface through different policy representations.

4.3 Why Battery Matters

Our goal is to select the "best" implementation. It is critical to come up with a rational as well as practical set of criteria to describe the "best". In order to solve this problem, we first need to answer a question: what do users care about most? We choose POMDP implementations in hope of improving mobile user experience. From the perspective of selecting a POMDP implementation, battery life is the top concerning issue. Assuming that we carefully select POMDP implementation candidates so that all potential choices provide the same level of service (i.e. all respond within a reasonable length of time with the same level of accuracy), the selection of POMDP implementations won't affect how users feel about the applications in short run. Because users don't care about how much CPU is utilized, neither do they care about how much memory is occupied (those implementations which use so much CPU or memory that affect the overall mobile experience are ruled out at the beginning and thus not considered during our selection). And no matter which POMDP implementation we choose, we always have to feed it with the same set of senor input and visualize the output in the same way. Therefore, the selection of POMDP implementations is transparent to mobile users in a short term. But in long run, users can feel the difference because all these CPU, WIFI, memory usage will drain the mobile battery. The mobile battery life is too noticeable to be ignored. In conclusion, as long as applications can provide similar level of service, e.g. respond with one or two seconds, can be loaded into memory without overwhelming other applications, WIFI or 3G works fine when needed, all that matters is battery consumption. From another point of view, the focus is also on battery consumption. According to smartphone user experience survey[8], battery life is among the top concerning issues of mobile experiences. The other top ones (such as signal strength, etc.) have nothing to do with whichever particular POMDP implementation we choose. And some survey even reveals that short battery life is the top one gripe from mobile users experience[25]. A POMDP application is supposed to

run for quite a long time, so the battery consumption is greater concern and even the most important factor when we consider backend POMDP implementations. Users will be extremely upset if our POMDP application kills their mobile battery within one or two hours.

4.4 Selection Criterion and Mobile Battery Consumption Model

Based on the above discussion, we decide to use battery consumption as our selection criterion. To be concrete, we use battery depletion rate as a POMDP implementations' battery performance indicator. The lower this depletion rate, the better its battery performance is. An intuitive way to measure this depletion rate is through real experiments. For each individual POMDP implementation, we run it for a while, and record the battery consumption. The battery depletion rate is calculated by battery consumption divided by time spent. However, this is a very inefficient way. This is because small amount of battery depletion won't provide statistically enough samples, and it takes hours for mobile devices to show noticeable battery depletion. This results in hours or even days of experiments. To avoid it, we will have to minimize the number of real battery experiments. As it turns out, mobile battery consumption consists of several bulk parts. We can estimate the battery depletion rate based on this knowledge.

According to a mobile battery study[5] conducted by researchers from University of New South Wales, the top four battery consuming components are: screen, signal standby, CPU usage and network (here we only consider WIFI) communication. Screen and signal standby are beyond our control. And also they don't matter for specific context because no matter what the internal POMDP implementation is, user experience (screen display) is the way it's supposed to be and signal standby is always there. Thus, in our design, CPU

usage and WIFI communication are the main factors we consider for battery consumption estimation.

We want to build a mobile battery consumption model that takes CPU usage and WIFI communication as arguments and outputs estimated battery depletion rate. Suppose the POMDP policy executes on discrete time interval T. In the implementation, it's essentially the policy executes in a infinite loop, during which the time length of each round is T. Obviously, executing one round every one second consumes differently from every ten seconds. Suppose the actual running time of each round is t within time interval T, and the rest $T-t$ is in idle. The estimated battery depletion rate can be estimated by following formula (4.1 is for using mobile only, 4.2 is for using client/server mode):

$$r = r_{\text{CPU}} \cdot t/T + r_{\text{Base}}(T) \tag{4.1}$$

$$r = r_{\text{WIFICom}} \cdot t/T + r_{\text{WIFIIdle}} \cdot (T-t)/T + r_{\text{Base}}(T) \tag{4.2}$$

We define benchmark as (r_{CPU}, r_{WIFICom}, r_{WIFIIdle}, r_{Base}). It can be obtained through real battery experiments on target device. One assumption is that, benchmark on similar device is the same, at least it won't affect final ranking of POMDP implementations that is based on estimated battery depletion rate.

After obtaining our benchmark result, for every new POMDP on similar devices, all we need to do is to measure the average execution time t during a given interval T. Then we can calculate the battery depletion rate based on above formula. The calculated battery depletion rate is our final criterion to rank implementations. In the final result sheet, we will rank the implementations and also list their memory usage and WIFI necessity as reference for users' ultimate decision. Even if an implementation ranks the top of saving battery, people may still decide to rule it out for the practical situation is not permitted with its memory usage or WIFI connection requirement.

Symbol	Explanation	How to obtain
$r_{\text{Base}}(T)$	Battery depleting rate that has the whole structure working but during each T, no querying action or updating belief state.	For each T, log timestamp of battery change, calculate average; eventually represented through regression or mapping table
r_{CPU}	Extra battery depleting rate cost by running full cycle computation, compared to mobile doing nothing.	r_{Zero} = battery depleting rate of device doing nothing; r_{WithCPU} = battery depleting rate of doing full CPU job; $r_{\text{CPU}} = r_{\text{WithCPU}} - r_{\text{Zero}}$
r_{WIFICom}	Extra battery depleting rate cost by intensive WIFI communication, compared to mobile doing nothing.	Similar as above
r_{WIFIIdle}	Extra battery depleting rate turning WIFI on but let it idle, compared to mobile doing nothing.	Similar as above

Table 4.1: Important Symbol Explanation

4.5 Real Experimental Battery Depletion Rate and Estimated Battery Depletion Rate

Now we have two ways to obtain battery depletion rate for each POMDP implementation candidate. One is to run real battery experiment for every POMDP implementation and log battery changes for several hours. At the end of each experiment, we use the total battery changes divided by experiment duration to get a average battery depletion rate. This battery depletion rate comes from real battery experiment and is therefore called real experimental battery depletion rate. Another way is to apply the mobile battery consump-

tion model described in Chapter 4.4. We first conduct a series of benchmark experiments, obtaining the results of (r_{CPU}, r_{WIFICom}, r_{WIFIIdle}, r_{Base}). This might take several hours. After that, we simply run each POMDP implementation for a couple minutes and record the average execution time for one round of POMDP execution. Then we feed the benchmark combined with average execution time into the mobile battery consumption model to generate an estimated battery depletion rate for every POMDP implementation. If our assumption in Chapter 4.4 holds, which means benchmark can be used across similar devices, we only need several minutes for each POMDP implementation to produce the estimated battery depletion rates as opposed to several hours for each POMDP implementation to obtain real experimental battery depletion rates. Even if the assumption does not hold and we have to obtain benchmark figures for every target device, we will still save a lot of time given sufficient amount of POMDP implementation candidates because the estimation method reduce experiment time from hours to minutes for every POMDP implementation candidate.

4.6 System Architecture

We design a software toolkit that automates POMDP implementation selection based on the battery depletion rate. As shown in Figure 4.2, our toolkit architecture contains three main components: input preparation, experiments and result analysis, invoked one after another.

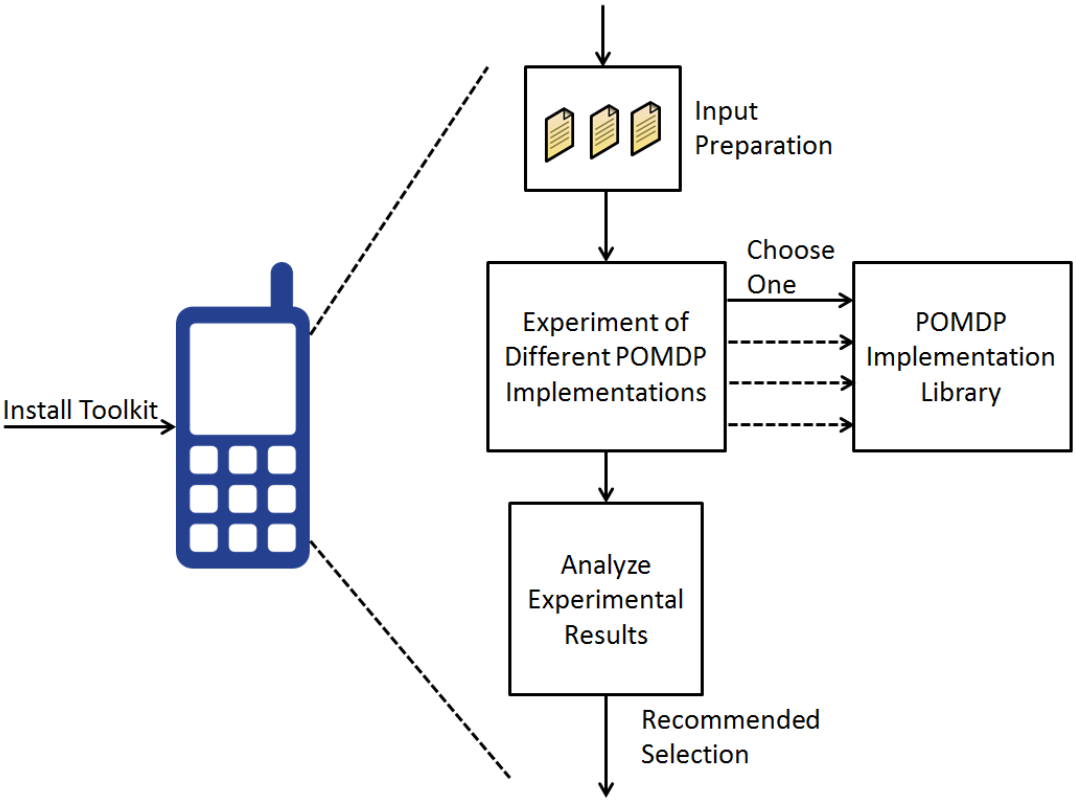

Figure 4.2: System Overall Architecture

4.6.1 Input Preparation

After installing our toolkit on a target mobile device, user can load input files down to target device. Input files consist of three parts:

1. Benchmark file. Since we make assumption in section 4.4 that same benchmark can be shared across similar devices, benchmark is an optional input. If not provided, new benchmark experiment has to be conducted on target device. Details of benchmark file are in Chapter 5.3.

2. POMDP problem description. The POMDP problem is expressed using the parentheses structure stated in Chapter 2.4. It contains all states, actions, observations, rewards information and their corresponding probabilistic data if any.

3. Policy files. Different implementations may execute on different policy files. The format of policy files are discussed in Chapter 2.3 and further addressed in Chapter 6.2.2.

4.6.2 Experiment

During the experiment part, our toolkit will first check whether the benchmark data are provided and acknowledged by users. If not provided or the user wants to update the benchmark data, a benchmark experiment is performed to gather basic battery depletion rate information of the target device, illustrate in Pseudo code 2. After we assure ourselves that benchmark data are usable, we can do experiment for each implementation or a selected set of candidates. Each implementation of interest will be brought up and executed continuously for certain amount of time. Through dividing execution time by number of rounds executed, we can calculate average execution time for one single round for one specific POMDP implementation. That is our execution time t in section 4.4. Now we have all we need to compute our estimated battery depletion rate, illustrated in Pseudo code 3. Finally, a sorted list based on estimated battery depletion rate is returned.

4.6.3 Result Analysis

The experiment component will return a sorted list of pairs of implementation and estimated battery depletion rate. Our final result is presented in the same order as the estimated battery depletion rate increases. The top one has the lowest estimated battery depletion rate, which means it is the most battery saving implementation. However, as

Pseudocode 1: Main Entry of Experiments

Input: $time\_intervals = \{T_1, T_2, ..., T_n\}$

Output: Ranking list for all implementation candadiates: $ranking[\,]$

```
if needRetestBenchmark( ) == True then
    benchmark = testBenchmark(time_intervals)
else
    benchmark = readBenchmarkFromInput( )
end
for implm in POMDP implementation library do
    t = averageExecutionTimeForOneRound(implm)
    ranking[implm] =
    estimatedBatteryDepletionRate(implm, t, time_intervals, benchmark)
end
sortOnRanking(ranking)
```

Pseudocode 2: Test Benchmark Function

Input: $time\_intervals = \{T_1, T_2, ..., T_n\}$

Output: *benchmark* information written to file

```
r_Zero = testBatteryDepletionRateForDoingNothing()
r_Base(T) = testBatterDepletionRateForIdlePOMDP(time_intervals)
r_CPU = testBatteryDepletionRateForFullCPU() − r_Zero
r_WIFICom = testBatteryDepletionRateForFullWIFICom() − r_Zero
r_WIFIIdle = testBatteryDepletionRateForWIFIIdle() − r_Zero
benchmark = (r_CPU, r_WIFICom, r_WIFIIdle, r_Base(T))
writeToFile(benchmark)
```

Pseudocode 3: Estimated Battery Depletion Rate

Input: Implementation: $implm$, Execution Time: t,
$time\_intervals = \{T_1, T_2, ..., T_n\}$, $benchmark$

Output: Estimated average battery depletion rate: r

$(r_{\text{CPU}}, r_{\text{WIFICom}}, r_{\text{WIFIIdle}}, r_{\text{Base}}(T)) = benchmark$
$result[\,]$
for $T \in time\_intervals$ **do**
    if $isUsingWIFI(implm)$ **then**
        $results[T] = r_{\text{WIFICom}} * t/T + r_{\text{WIFIIdle}} * (T - t)/T + r_{\text{Base}}(T)$
    else
        $results[T] = r_{\text{CPU}} * t/T + r_{\text{Base}}(T)$
    end
end
$r = average(result[\,])$

pointed out at the end of section 4.4, we also need other auxiliary information to better assist users' decision making. We provide statistics about peak memory usage and WIFI necessity of implementations. Therefore, the displayed final result is a list of 4-tuples: (Implm, Ranking, Mm Use, WIFI Req), ordered by ranking.

Chapter 5

Implementation

We choose to implement our toolkit on Android platform. There are three main components in our design: (1) a framework that controls and organizes all experiments; (2) an experiment component dedicated to test individual POMDP implementation, which may be invoked by the framework; and (3) a result analyzing component which takes all the experimental results and displays them in a user understandable format.

On Android platform, the concept of activity is used to organize independent functional logic. An application has one or more activities depending on how a developer wants to organize its structural unit. We implement each of the three components as one single activity. This is not only because we want to decouple the system and maintain a clear structure, but also because we want to give users more control over each part of the experiment process. For example, if a user wants to examine one particular POMDP implementation and check its battery performance, he has the option to manually launch the experiment activity and specify parameters for that particular POMDP, and then open the result analyzing activity after the experiment finishes.

The following Figure 5.1 identifies two different paths of using the toolkit.

1. Launch framework activity and let it handle everything. It will first decide whether we need to re-launch benchmark activity. If needed, launch it and get the benchmark results. Then for a POMDP implementation candidate set, launch experiment activity one for each candidate. In the end, launch result analyzing activity and show final result.

2. No need of management from framework activity. User can control whether they want to do benchmark test again. And user can also decide which POMDP implementation they are interested in and launch an experiment activity specifically for it. In addition, user can choose when to open the result analyzing activity to display result of those interested implementations.

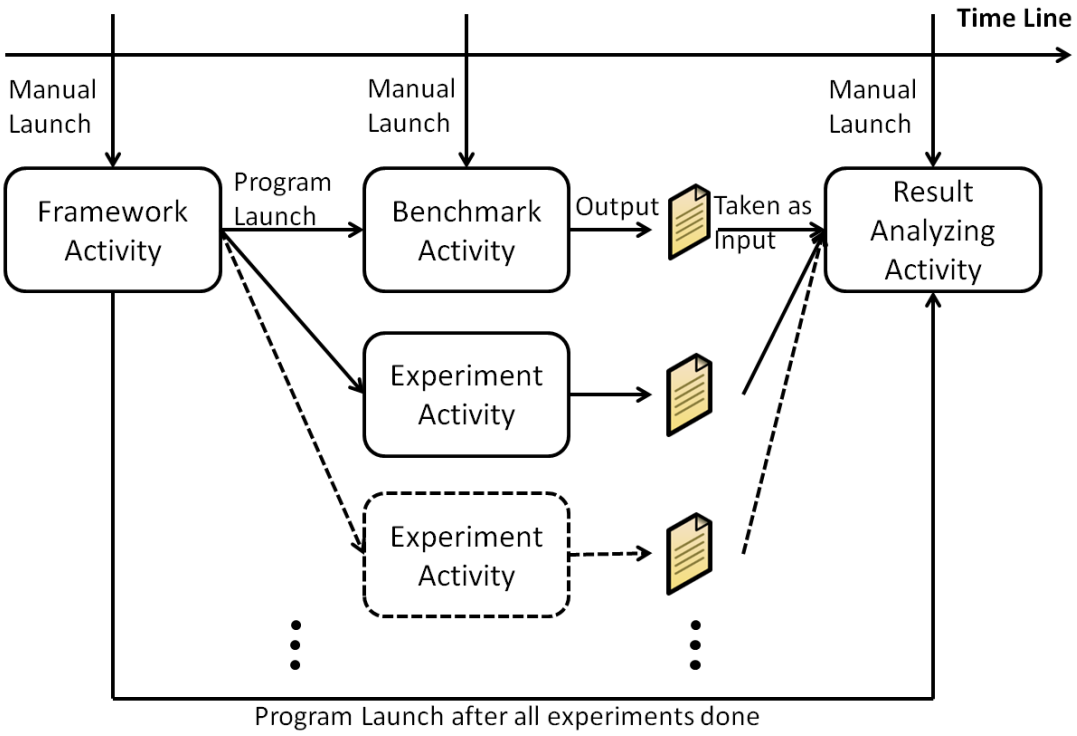

Figure 5.1: Android Activities Organization

5.1 Activity Communications

Since we want to logically separate activities and allow them to work independently, the information that needed to be passed among activities should be stored in files. The following table illustrates dependencies among activities.

Activity	**Input**	**Output**
Framework Activity	Configuration	Graphical display of output from Result Analyzing activity
Benchmark Activity	None	$r_{\text{Base}}(T)$, r_{CPU}, r_{WIFICom}, r_{WIFIIdle}
Experiment Activity	Implementation choice	Execution time per round t for the chosen implementation and corresponding memory usage, WIFI necessity.
Result Analyzing Activity	Output of benchmark activity, all the output of experiment activity	Final ranking of implementations

Table 5.1: Activities Dependency

According to the table above, information that needs to be logged to provide interactivity communication is: output of benchmark activity; output of each experiment activity instance; output of result analyzing activity. They are all simple information which can be stored in plain text files, as shown in Figure 5.2.

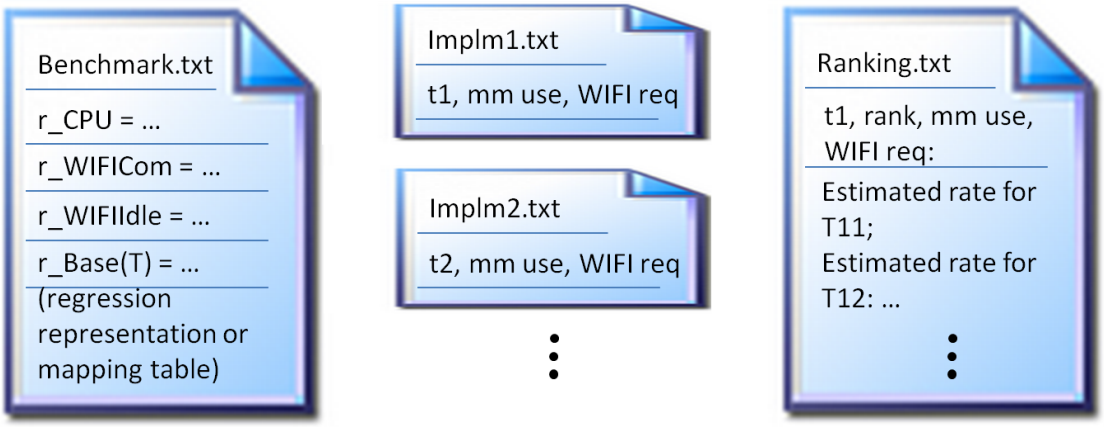

Figure 5.2: Information Representation

Each time when one activity needs input, it just needs to read the corresponding files. Configuration file is a global setting file that every activity can access, specified in next section.

5.2 Framework Activity

Framework activity's job is critical but also simple. It launches benchmark activity; and then, several experiment activities; in the end, the result analyzing activity to show the recommended ranking. The whole process is based on the following configuration, as shown in Table 5.2.

relaunching_benchmark_needed configuration controls whether we need to launch the benchmark activity. If we already launched benchmark activity not long ago, or we get benchmark result from somewhere else that we are comfortable with, we don't need to re-launch it every time we want to examine some implementations, as it is time-consuming, discussed in section 5.3.

Configuration	**Value**
relaunching_benchmark_needed	True/False
implementation_candidates	$\{implm_1, implm_2, , implm_n\}$
time_intervals	$\{T_1, T_2, , T_n\}$

Table 5.2: Configuration Content

implementation_candidates specifies which POMDP implementations we want to examine. We can eliminate unwanted implementation to save time and provide a more focused ranking.

time_intervals specifies what time interval we use to test benchmark and also to calculate the average ranking for each POMDP implementation.

5.3 Benchmark Activity

Benchmark activity is expected to run quite a while for it has to perform real battery test. And since short term running (within an hour) won't yield a noticeable battery change, the experiment time for one single of r_{CPU}, r_{WIFICom}, r_{WIFIIdle}, $r_{\text{Base}}(T)$ might take hours. The way we measure battery depletion rate is by first recording timestamp and battery level logs: (ts_1, bl_1), (ts_2, bl_2), ..., (ts_n, bl_n); then calculate individual rate of each time period $r_i = (bl_i - bl_{i+1})/(ts_{i+1} - ts_i)$; in the end, average all rate to output a final rate for battery depletion.

Aside from how to measure battery depletion rate, another equally important question is that on what settings we obtain the battery depletion rates of r_{CPU}, r_{WIFICom}, r_{WIFIIdle}, $r_{\text{Base}}(T)$.

r_{CPU}, r_{WIFICom} and r_{WIFIIdle} are straightforward. First of all, we need to test r_{Zero}, which is the battery depletion rate with fixed screen brightness, WIFI off, as few other

applications running as possible. For r_{CPU}, we measure r_{WithCPU} from logging one simple application doing computation in a infinite loop (calculating the first 100 Fibonacci numbers) to simulate full cycle CPU usage, and then calculate r_{CPU} by $r_{\text{CPU}} = r_{\text{WithCPU}} - r_{\text{Zero}}$. Similarly, $r_{\text{WithWIFICom}}$ is a simple application doing communication with a remote server through WIFI in a infinite loop (sending a random integer and receiving a random integer to and from the server in each round to simulate full WIFI usage), and $r_{\text{WIFICom}} = r_{\text{WithWIFICom}} - r_{\text{Zero}}$. And the settings of $r_{\text{WithWIFIIdle}}$ is the same as r_{Zero} but with WIFI turned on, $r_{\text{WIFIIdle}} = r_{\text{WithWIFIIdle}} - r_{\text{Zero}}$.

For $r_{\text{Base}}(T)$, it is a little complicated. In order to be close to real performance, all r_{Base} measurement should be done as if on real POMDP execution. So, we introduce our POMDP execution simulation framework, as shown in Figure 5.3.

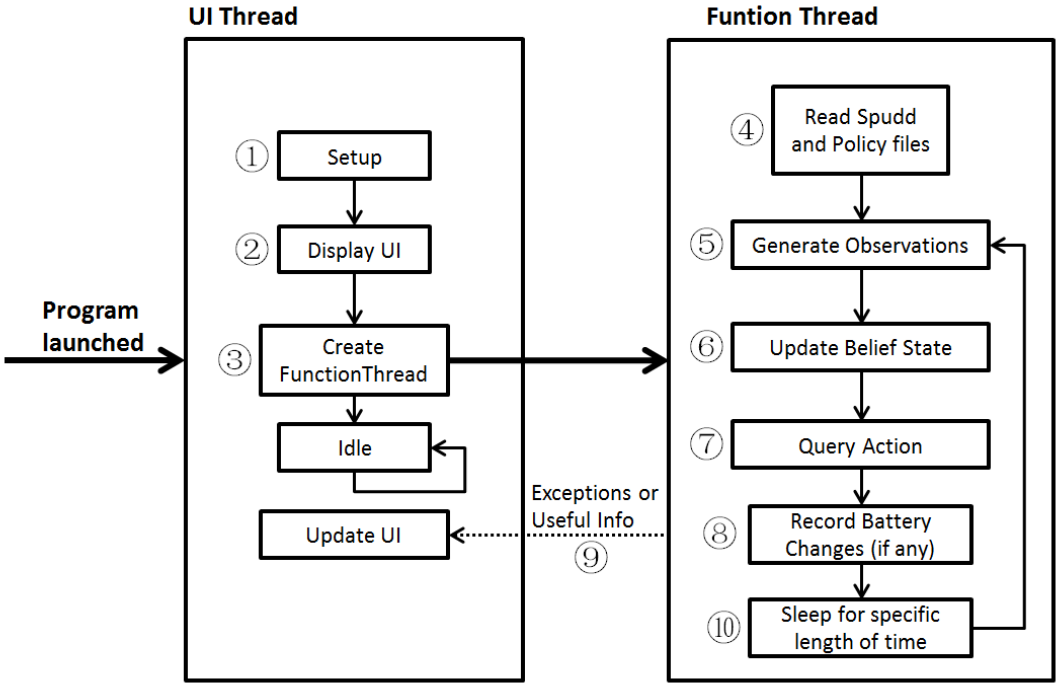

Figure 5.3: Program Details

(1) Basic setup for the experiment; includes reading configuration files, registering UI handlers and battery listener.

(2) Display user-interface to allow further configuration input.

(3) UI Thread creates a function thread that handles all POMDP related operations and records battery changes. After the creation, UI thread enters idle stage unless it is resumed by function thread for updating UI.

(4) First step of function thread is to read in all the necessary files for executing POMDP, e.g. POMDP specification files: SPUDD file and policy file if any (benchmark experiment doesn't need POMDP specification, but the program structure is meant to serve general POMDP execution purpose).

(5) Observations should come from sensors on mobile reflecting real environment changes. However, in order to simplify the process and avoid instability from unpredictable environment, we have our observations come from a random generator simulated in mobile application level. This is acceptable because as we discussed before, no matter which implementation we choose, we always have to feed it with the same sensor input. Thus, the sensors set can be simulated by software.

(6) Update belief state with regards to different POMDP implementation.

(7) Query action with regards to different implementations.

(8) A Battery listener is registered in step (1) and exists throughout the whole experiment. Every time when the battery level changes, Android OS will issue a broadcast which can be caught by battery listener to extract battery level information. The battery listener is independent from function thread. It will write battery log (time stamp and current battery level) to disk once notified. Battery changing notification is raised with granularity of 1%.

⑨ WIFI exception, IO exception and useful information such as average query execution time, peak memory usage will be displayed on UI.

⑩ Sleep for the rest of time interval and go back to start another round after the current time interval finished.

For ④, ⑥, ⑦, function thread probes into POMDP module to invoke corresponding methods respected to different implementation. POMDP module is a controller of POMDP implementation instance, providing interfaces of belief state maintenance and optimal action calculation. If runs in mobile-only mode, POMDP module instantiates POMDP execution locally on mobile. Otherwise (client/server mode), POMDP module will establish WIFI connection to update belief state and query actions on a remote server. The above structure not only serves for benchmark experiment, but also illustrates a framework for real POMDP execution.

For $r_{\text{Base}}(T)$ in benchmark, we can simply replace ⑥, ⑦ with a simple function just returning a void value rather than really trying to query action and update belief state.

We can have a set of base line battery depletion rate (r_{Base1}, T_1), (r_{Base2}, T_2) In the end, we can either represent $r_{\text{Base}}(T)$ as a value lookup table or use some regression.

5.4 Experiment Activity

The experiment activity utilizes the same POMDP execution framework as above. In each round, querying action and updating belief state act on corresponding entities. Currently, we have four POMDP implementations in library, as shown in Table 5.3.

All of the above implementations except Client/Server run totally on mobile phone. For the Client/Server, the whole POMDP executes on Server. Mobile first obtains observations

Implementation	Explanation
FSC	Take observations from generator and return actions based on a finite state controller.
Client/Server	Take observations from generator and send to a server through WIFI; Server runs Symbolic Perseus and responses to action query.
Symbolic Perseus	Take observations from observation generator and return actions based on Symbolic Perseus[27] method.
Flat Policy	Take observations from generator and return actions based on α-vector policy calculated by enumeration algorithm[22].

Table 5.3: Implementation Candidates Illustration

from its own sensors and then acts like a client to update belief states in server, followed by querying actions. The purpose of the experiment activity is to record the real execution time during each round, namely the time of updating belief state and querying action. These can be easy done with android API. After certain rounds, the average execution time is logged into file: *implmX.txt*. Also the memory usage and whether this implementation needs WIFI are also logged together.

5.5 Result Analyzing Activity

The result analyzing activity is the simplest one. It only needs to do some simple calculation given the data from files. For each POMDP implementation candidate, it calculates its final battery depletion rate average on its performances on all the time intervals, and use

it as the rank of this implementation. After computing all the ranks, result analyzing activity ranks all candidates based on their ranks, the smallest ranks the first. In the end, it displays an ordered result list that contains tuples (implm, ranking, mm use, WIFI req).

5.6 User Interface

There are three types of activities: Framework Activity, Benchmark/Experiment Activity and Result Analyzing Activity. We give them each a main user interface. For Framework Activity, we provide choices of what experiments the user wants to launch, including the benchmark test. A list of potential implementations are displayed for multi-selecting. After selecting their interested implementation, and pressing the Start experiments button, user interface switches to benchmark/experiment activity. During this activity, all the experiments are happening one by one in the background. The user can monitor runtime information from the two panels displayed (one for normal information, another for exception information). The user can also choose to temporarily stop the experiment activity whenever they want by pressing the Pause button. After each single experiment, if battery level is detected to be lower than 50%, a message box will pop up to prompt for re-charging in order to achieve comparable results. During this situation, the user only needs to connect device with its charger, and wait for it to be back to above 90%, then press Resume button to continue from what it left off before. After all selected experiments are performed; user interface will again switch to another display, showing the analysis of what it just ran. The whole process is visible and controllable by the user.

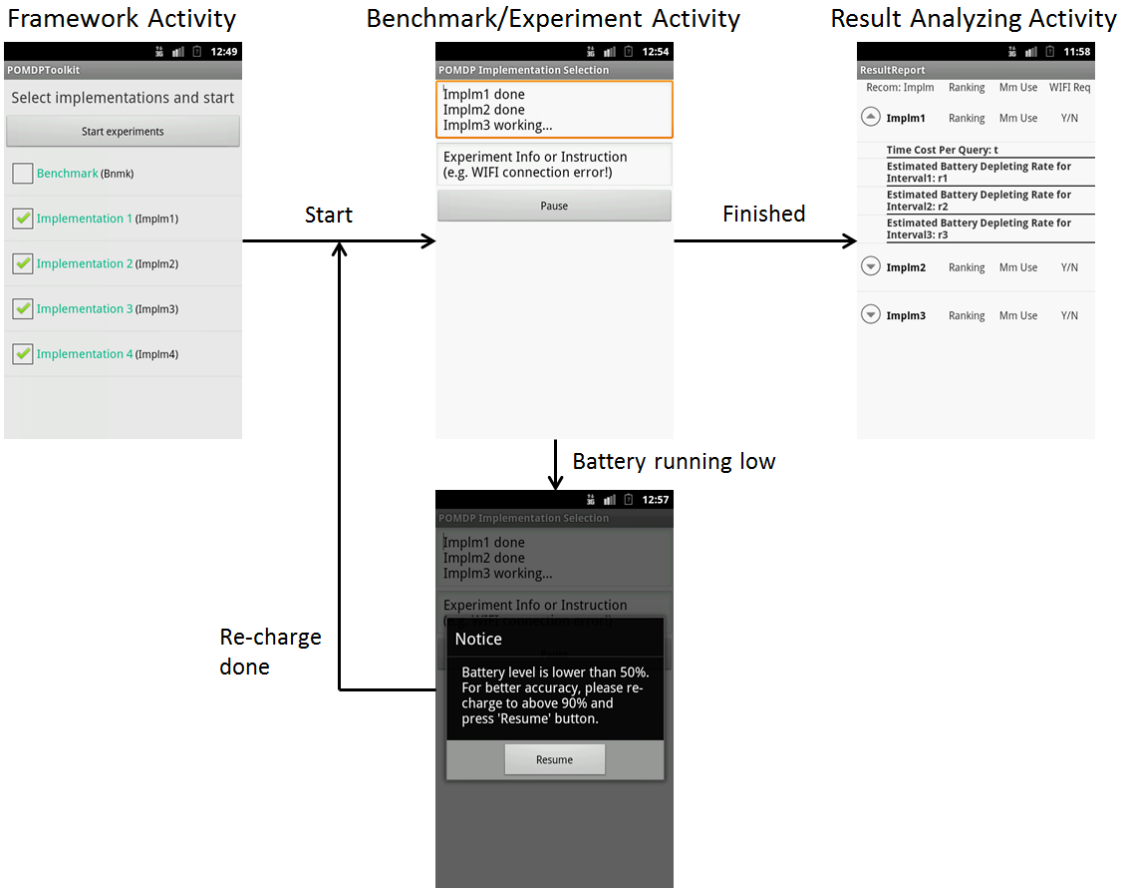

Figure 5.4: Graphical User Interface Examples

Chapter 6

Evaluations

We evaluate our work through comparisons between estimated battery depletion rates and those from real battery experiments. We first obtain the benchmark data (r_{CPU}, r_{WIFICom}, $r_{WIFIIdel}$, r_{Base}) by running benchmark experiment, as illustrated in Chapter 5.3. Then for each POMDP implementation candidate, we record the average execution time of one round, and apply them to Equation 4.1 and 4.2. Now, we have estimated battery depletion rates for all POMDP implementation candidates. Next, we gather battery depletion rates for all implementation candidates in real battery experiment. We use the program structure in Chapter 5.3 to log battery changes, and calculate an average battery depletion rate for each POMDP implementation candidate.

The expected outcome is that the ranking derived from estimated battery depletion rates is the same with one derived from real battery depletion rates. Better, if estimated battery depletion rates are close to real ones.

6.1 Evaluation Settings

6.1.1 Mobile Setting

All the experiments run on the same mobile device (Nexus 4, Android 4.2) to eliminate discrepancy caused by different devices. Each experiment starts with battery level above 95% and keeps running for about 3 hours, during which battery changes are recorded as files on mobile storage.

We try to shut down as many unrelated user applications as possible. No SIM card is installed on the smartphone for it is unnecessary and also avoid uncontrollable interference (usually depends on too many factors) from battery consumption of signal standby. WIFI is turned on only when necessary ($r_{WithWIFI}$ and $r_{\text{WithWIFIIdle}}$ test, and Client/Server implementation experiment).

The screen is on and set to fixed brightness (50% brightness) throughout every three-hour experiment for two reasons: 1), Android OS treats screen-off as a signal of low usage, as it may slow down CPU and disconnect WIFI, which is unquantifiable and consequently diminishes the meaning of comparison among different implementations; 2), If the screen is off, the smartphone enters battery saving mode. Hours (more than 10 hours) of experiment will only consume a very small amount of battery (about 3-10%). Statistically, this is inaccurate for we don't have sufficient battery changes to calculate good average battery consumption rate.

6.1.2 POMDP Setting

Every experiment deals with the same problem setup - Location and Context Aware Safety Assistant. It has 2880 states, 72 observations, and 6 actions. We run an infinite horizon POMDP.

FSC: The finite state controller. The POMDP policy is captured by some internal states that the execution is essentially updating states given observation and returning prescribed actions. It has 321 internal nodes. The implementation maintains internal states in two dimensional array, within which each row represents the connections and actions associated. The state updating is just table lookup and index updating with minimum computational complexity.

Symbolic Perseus: The implementation executes based on a α-vector policy. During each query, it searches for the best α-vector that maximizes future rewards given the updated belief states. In our experiments, the α-vector policy contains 26 α-vectors.

Client/Server: Client side reads the POMDP description, server side executes on the same Symbolic Perseus as above. In fact, it doesn't matter what implementation is the on server side because we can assume all implementations response immediately for our relatively small problem. And data transmitted are simple state and action expression. Thus, neither the WIFI waiting time nor data transmissions differ a lot among different POMDP methods on the server.

Flat Policy: Similar to Symbolic Perseus implementation, but it's a simplified version of flat structure of implementation of α-vector policy execution.

6.1.3 Time Interval Setting

Apparently, different execution frequency should have various impacts on battery decreasing, e.g. executing observing, querying, and updating in every 10 seconds consumes much less than executing them in every second. Therefore, the time interval is an important parameter in our experiment. The program runs in a infinite loop. We set up different benchmarks in terms of time intervals between each individual round of executions: 10 seconds, 2 seconds, and 1 second. During each round, the program obtains observations,

updates its belief state and queries action, and then sleeps for the rest of the interval. For example, if it takes 0.1 second to finish the above series of executions, the program will then sleep 9.9 seconds for the 10 seconds interval experiment, 1.9 seconds in 2 second interval experiment, and so on.

6.1.4 Battery Logging Method

In Android OS, the method we access battery information is to register a power management class that will listen to system broadcast. Every percentage of battery change will trigger a system level notification that is passed to the upper level. The power management class can catch this information and update on application side. Every time the battery changes, we log the current battery level associated with a time stamp. The record has format $(batteryLevel, timeStamp)$, (bl, ts) for short. We record every percentage of battery changes and calculate the average battery depletion rate from using $r = (bl_i - bl_{i+1})/(ts_{i+1} - ts_i))$.

6.2 Results

We first examine the results from benchmark experiments, and then we obtain necessary data to produce the estimated the battery depletion rate. After that, we compare the battery depletion rates from pure experiments and estimation. We focus on the ranking and also the actual number of battery depletion rates. In the end, we display the suggestion of POMDP implementations based on estimated battery depletion rates.

6.2.1 Benchmark

We conduct the benchmark experiments based on the program structure illustrated in Chapter 5.3. By setting the *updateBeliefState* function and *queryAction* function as void function (simply and only returning void), we obtains r_{Base} for three different time interval settings (10 sec, 2 sec, 1 sec), as shown in Table 6.1. To obtain other parameters in the benchmark, we followed the description in Table 4.1. First, we experiment on r_{Zero}, which is the battery depletion rate of unavoidable energy consumption of keeping the mobile alive (fixed screen brightness, WIFI turned off, as many as possible other applications closed). This r_{Zero}, as stated in Chapter 4.4, is used to calculate battery consumption of pure CPU usage and WIFI related usage. Then, we test on r_{WithCPU}. It is the battery depletion rate of having a Fibonacci number calculation in a infinite loop to simulate full cycle CPU usage on mobile device. We compute the battery depletion rate of pure CPU usage by $r_{\text{CPU}} = r_{\text{WithCPU}} - r_{\text{Zero}}$, which is 0.150. Also, we run experiments of the battery depletion rate of full WIFI communication (mobile continuously sends and receives a random number to or from a server through WIFI) and WIFI idle (same status as r_{Zero} experiments but with WIFI turned on). Then we compute benchmark $r_{\text{WIFICom}} = r_{\text{WithWIFICom}} - r_{\text{Zero}}$ (0.025), and $r_{\text{WithWIFIIdle}} = r_{\text{WithWIFIIdle}} - r_{\text{Zero}}$ (0.001). The average battery depletion rates are all calculated from 40 samples.

Benchmark	Results		
	Time Interval (sec)	Battery Depletion Rate (%/min)	Standard Error
$r_{\text{Base}}(T)$	10	0.158	0.0087
	2	0.157	0.0061
	1	0.172	0.0033
r_{Zero}	N/A[1]	0.148	0.0039
r_{WithCPU}	N/A	0.298	0.0132
$r_{\text{WithWIFICom}}$	N/A	0.173	0.0071
$r_{\text{WithWIFIIdle}}$	N/A	0.149	0.0035

Table 6.1: Benchmark Results

6.2.2 Real Experiments

By applying the program structure from Chapter 5.3, we obtain results of four typical POMDP implementation, as shown in Table 6.2. The tested results are the results from real battery experiments. The estimated results are calculated as illustrated in Chapter 4.4, and therefore doesn't have standard error data associated with it. Our intention is to compare the difference between tested results and estimated results, mainly about the ranking difference derived from these two results. It is also interesting to see how close these two sets of numbers actually are. The avg exe time stands for the average time cost in one single round for executing all POMDP necessary operations, which is used in Chapter 4.4 to calculate the estimated results. For example, in Symbolic Perseus, they are observation gathering, belief state updating and action querying. For Client/Server, it is gathering observation and contacting server for the next action, while for FSC, it is

[1]The idea of time interval is not applicable here because the mobile device is running continuously without periodical break.

gathering observation and following the state changes of the internal controller.

Implementation	**Tested Results**			**Estimated Results**		**Avg Exe Time**
	T[2]	**BDR**[3]	**StdE**[4]	**T**	**BDR**	
FSC	10	0.159	0.0047	10	0.158	0.001 sec
	2	0.154	0.0045	2	0.157	
	1	0.175	0.0087	1	0.172	
Client/Server	10	0.166	0.0064	10	0.161	0.898 sec
	2	0.172	0.0076	2	0.169	
	1	0.181	0.0090	1	0.194	
Symbolic Perseus	10	0.176	0.0049	10	0.168	0.669 sec
	2	0.226	0.0058	2	0.207	
	1	0.256	0.0111	1	0.272	
Flat Policy	10	0.166	0.0041	10	0.165	0.472 sec
	2	0.207	0.0060	2	0.192	
	1	0.244	0.0085	1	0.243	

Table 6.2: Implementation Results

6.2.3 Comparison for Evaluation

We draw the comparison graph below. The estimated results are basically aligned with real results (Figure 6.1). And ranking derived from tested result is exactly the same with that from estimated result (Figure 6.2). We also compare tested results and estimated results by computing cosine similarity:

[2]Time Interval (sec)
[3]Battery Depletion Rate (%/min)
[4]Standard Error

$$similarity(A, B) = cos(A, B) = \frac{A \cdot B}{\|A\| \cdot \|B\|} = \frac{\sum_{i=1}^{n} A_i \cdot B_i}{\sqrt{\sum_{i=1}^{n} (A_i)^2} \cdot \sqrt{\sum_{i=1}^{n} (B_i)^2}} \tag{6.1}$$

The cosine similarity ranges from 0 to 1 where 1 indicates they are extremely similarity. The cosine similarity of our tested battery depletion rate and estimated battery depletion rate is 0.999, 0.998, 0.997 and 0.999 respectively for FSC, Client/Server, Symbolic Perseus and Flat Policy, which means our estimation well predicts the real performance.

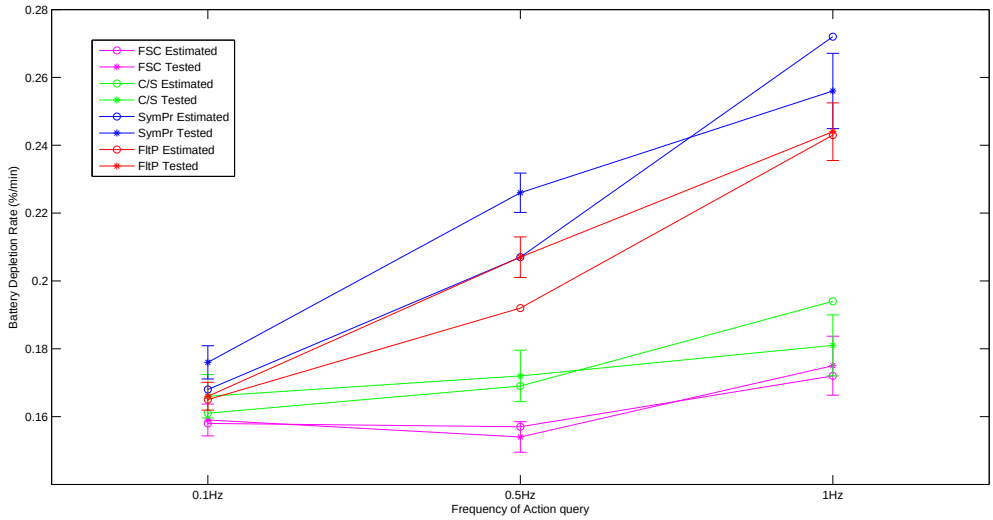

Figure 6.1: Each data point in the figure is the battery depletion rate of corresponding implementation in given frequency of action query. The estimated battery depletion rate doesn't have standard error associated with it because it is obtained through estimation and analysis method mentioned in Chapter 4.4, rather than pure battery experiments.

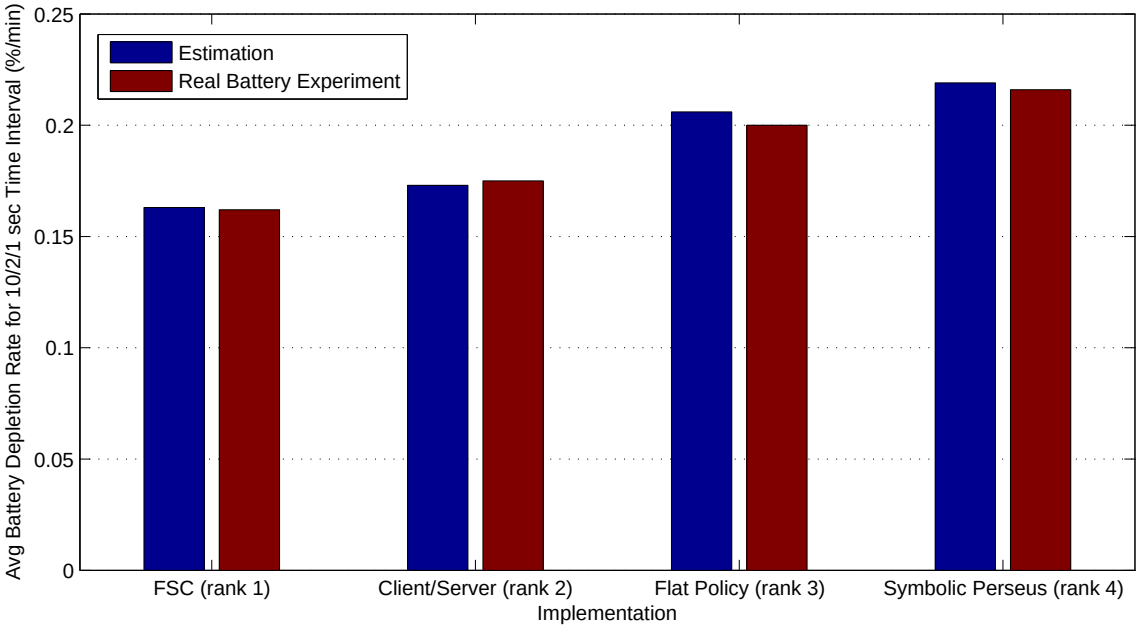

Figure 6.2: Each data point in the above figure is the average battery depletion rate of corresponding implementation on time interval 10 sec, 2 sec, and 1 sec. We use the average battery depletion rate to derive a ranking of implementations. Obviously, the lower the average battery depletion rate, the better one implementation ranks. The above figure also illustrates that the ranking derived from our estimation real battery experiments are the same, which justifies the correctness of our estimation. Though there are discrepancies in actual numbers, our ultimate goal is to compose a ranking of implementations.

6.3 Implementations Selection Suggestion

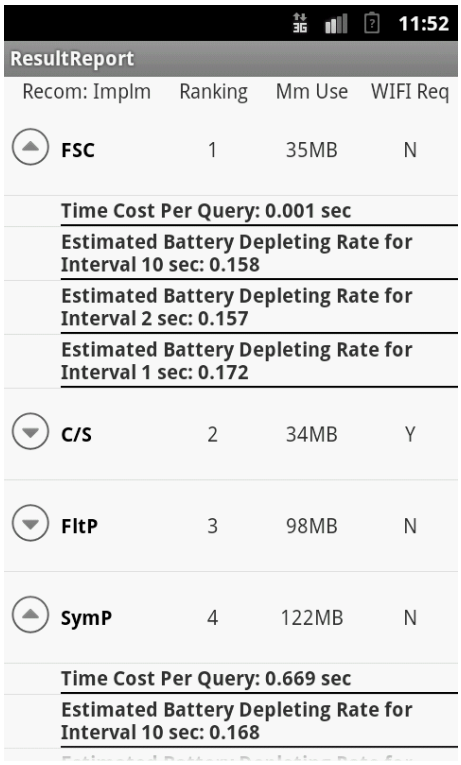

Figure 6.3: Final Ranking Visualization

The suggestion is displayed on a result reporting sheet above on mobile device (Figure 6.3). FSC made the first place with slowest battery depletion rate, reasonable amount of memory usage and no requirement of WIFI communication. What follows are client/server, flat policy and Symbolic Perseus. Our ranking is first based on battery depletion rate. If the same, we look at whether the implementations need WIFI communication, the one without such requirement ranks higher since it introduces less restriction. At last, we compare memory usage. The one with less memory usage is better.

Critical readers may argue that FSC is internally different from other three. Every

action query is just table lookup with minimum computation complexity, and it's therefore better in energy saving [14]. As there are technologies that enable controller search and translation from other policies to finite controller policies [12, 13], the FSC policies would be the default preference, obviating the need for battery experiments.

The argument presented in this thesis goes beyond judging which implementation is best. It's more about providing a way such that whenever we want to assess the implementations, we have a convenient and trustworthy tool that can automate the evaluation process. The purpose of this work is to enlarge the POMDP application domain and increase its popularity among a large number of mobile developers, not just POMDP researchers and experts. Think about a scenario where mobile developers are facing an upcoming deadline of a POMDP project and they have little idea of POMDP's theoretical models. They have some available implementation packages and some already-computed policies. They would appreciate a tool that can help them make quick and right selection from the available choices. This is the motivation of our work. This is the value of our work. Besides, we are also targeting the bigger picture. We foresee that our work can be generalized for other artificial intelligence and machine learning models.

Chapter 7

Conclusion

7.1 Discussion

In this thesis, we pointed out selecting proper POMDP implementations for mobile devices is actually much more important than people have thought about. And the key question is how to design selection criteria to select POMDP implementations so that we can make ranking of implementation candidates. We construct our selection criteria based on mobile battery consumption of implementation candidates and rank them based on their battery depletion rates. We developed a framework dedicated to measure applications' mobile battery performance, mainly but not exclusively for POMDP applications. We then accelerated the whole measuring process by introducing our mobile battery consumption model, which uses the execution time of full cycle CPU and WIFI communication to approximate battery depletion rate of running a particular applications. At last, we built a software toolkit that automates the whole process. It is so simple that all users need is to install the toolkit, provide POMDP specification files, launch the toolkit, and wait for a comprehensive suggestion of which POMDP implementation candidate to choose.

This thesis has bigger meaning beyond the selection of a good POMDP implementation. Through the study of implementation selection, it reveals that the implementation of a computational heavy model on mobile platform should not solely depend on intuition. There are actual methods to rationalize the selection. And it does not just apply to POMDP. Other similar Artificial Intelligence and Machine Learning models may have the same potential to improve their user experience on mobile platform.

7.2 Future Work

We made an assumption in Section 4 that using same benchmark results among similar devices would produce the same ranking. Even if it's different, the difference won't matter a lot. This seems like a reasonable assumption, but still needs some experiments to support it. In the future, we plan to test it on multiple devices to see if that assumption holds.

As for the experiments in our work, currently it is one time experiment for one implementation, during which we record time stamp for every one-percent battery depletion. We are interested in examining larger amount of experiments to generate convincing results. Now the POMDP implementations are library provided by us. We want to upgrade the toolkit so that it can provide more general interface for third party POMDP libraries, so that it has more flexibility.

The design is only for POMDP applications. But we think a lot of other Artificial Intelligence or Machine Learning applications share similar features have space for more battery saving version. We hope we can generalize our work so that it can be extended to assist the application development of other fields.

We are dong health informatics related researches. The results shouldn't be buried in the lab. We are looking forward to coordinate with other institutions to apply it to realistic scenarios.

APPENDICES

Appendix A

Symbol Explanation

Symbol	Explanation
bl	Battery level with unit of percentage, indicating how many percentage of battery left
ts	Time stamp indicating the current system time when one battery level is logged
T	Time interval of one round POMDP execution
t	Actual running time during each round of time interval T
r_{Zero}	Battery depletion rate of fixing screen brightness, turning of WIFI, turning of as many applications as possible
r_{WithCPU}	Battery depletion rate of calculating Fibonacci number in a infinite loop (simulating full cycle CPU usage)
$r_{\text{WithWIFICom}}$	Battery depletion rate of continuously sending random integer and receiving random integer to and from a server through WIFI
$r_{\text{WithWIFIIdle}}$	Battery Depletion rate of simply turning on WIFI but not using it at all

$r_{\text{Base}}(T)$	Battery depletion rate that has the whole structure working but during each T, no querying action or updating belief state
r_{CPU}	Extra battery depletion rate cost by running full cycle computation, compared to mobile doing nothing, calculated by $r_{\text{WithCPU}} r_{\text{Zero}}$
r_{WIFICom}	Extra battery depletion rate cost by intensive WIFI communication, compared to mobile doing nothing, calculated by $r_{\text{WithWIFICom}} r_{\text{Zero}}$
r_{WIFIIdle}	Extra battery depletion rate turning WIFI on but let it idle, compared to mobile doing nothing, calculated by $r_{\text{WithWIFIIdle}} r_{\text{Zero}}$

Table A.1: Symbol Explanation

Appendix B

Smart Tasks Division between Mobile and Server to Save Mobile Battery

Mobile Cloud Computing (MCC)[19], which intends to combine mobile devices and cloud computing, is an increasingly hot topic nowadays. Researchers are trying to take advantages of both superiority to explore a better way of providing services. Mobile devices are portable and cheap while the cloud is powerful and centralized. Traditional cloud computing suggests that we put everything in the cloud; however, the strong growth of performance of mobile devices indicates that we may want to consider having some tasks remained on mobile devices. That posts an interesting question: how should we divide tasks between mobile and Server? One particular angle to tackle this problem is to make division based on mobile battery consumption, since it is the most visible and concerning problem in mobile user experience[8, 25].

We define a program block as three steps: *read*, *calculate* and *write* in order. Every program can be abstracted as sequence of blocks one after another. To specifically address the cooperation between mobile devices and the server, we further split the concept of each step based on where they execute - either "on mobile" or "on server". The three steps

read	***calculate***	***write***	**Mobile CPU Usage**	**Mobile Network Usage**
read_on_mobile	*calculate_on_mobile*	*write_on_mobile*	Calculation	Nothing
read_on_server	*calculate_on_mobile*	*write_on_mobile*	Calculation	Receive input
read_on_mobile	*calculate_on_server*	*write_on_mobile*	Nothing	Send input and receive output
read_on_mobile	*calculate_on_mobile*	*write_on_server*	Calculation	Send output
read_on_server	*calculate_on_server*	*write_on_mobile*	Nothing	Receive output
read_on_server	*calculate_on_mobile*	*write_on_server*	Calculation	Receive input and send output
read_on_mobile	*calculate_on_server*	*write_on_server*	Nothing	Send input
read_on_server	*calculate_on_server*	*write_on_server*	Nothing	Nothing

Table B.1: Mobile Behaviors according to Choice of Locations Where *read*, *calculate* and *write* execute

become *read_on_mobile* or *read_on_server*, *calculate_on_mobile* or *calculate_on_server*, and *write_on_mobile* or *write_on_server*. The following Table B.1 illustrates how mobile CPU and network component perform according to different location - either "on mobile" or "on server" - for each step. We only consider battery cost of CPU and WIFI usage since they are the main factors that are related to tasks division[5].

We then quantify the battery consumption of CPU calculation, and network communication based on experimental results of battery tests, which results in a co-relation between the tasks processed and battery depletion rate of mobile devices. It is done by conducting

experiments on various test cases (different amount of data, different complexity), and use regression to approximate the co-relation. Prior to the implementation of the program, the developers should be able to provide information about the tasks, mainly about the blueprint of blocks structure as well as amount of data and complexity involved (e.g., 10K data, $O(n^2)$ time complexity for $block_i$). This information is fed into our co-relation (e.g., $cost_{CPU} = f_1(10K, O(n^2))$, $cost_{WIFI} = f_2(10K)$, f_1 and f_2 are functions we obtain in battery experiments) to come up with an estimation of mobile battery cost of a given block. If the developer can provide amount of data processed and time complexity associated for all blocks, we can conclude on a overall cost. Choices can be made for each block between executing on mobile or executing on server. Difference choices result in different mobile battery costs. For example, if one block is executed on server, there will be not $cost_{CPU}$ for this particular block on mobile because the *cost* only captures battery consumption on mobile since we only care about mobile battery consumption. But it potentially causes more network communication usage since data that are originally on mobile need to be sent to server now. Many more trade-offs can be made during the selection of execution locations. By intelligently identifying the execution location of every block, we compute an minimized overall cost and an optimal schema of tasks' execution location associated with it in terms of saving the most energy. This optimal schema can be used as reference when actually implementing the program. For details of this project, please refer to [41].

References

[1] Cesar Bandera, Francisco J Vico, José Manuel Bravo, Mance E Harmon, and Leemon C Baird. Residual q-learning applied to visual attention. In *ICML*, pages 20–27, 1996.

[2] Darius Braziunas. Pomdp solution methods. *University of Toronto, Tech. Rep*, 2003.

[3] Edward Brorein. New tools for optimizing operating time of mobile wireless devices. Agilent Technologies, 2002.

[4] Ed Burnette. *Hello, Android: introducing Google's mobile development platform*. Pragmatic Bookshelf, 2009.

[5] Aaron Carroll and Gernot Heiser. An analysis of power consumption in a smartphone. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, pages 21–21, 2010.

[6] Anthony R Cassandra. A survey of pomdp applications. In *Working Notes of AAAI 1998 Fall Symposium on Planning with Partially Observable Markov Decision Processes*, pages 17–24, 1998.

[7] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. Clonecloud: elastic execution between mobile device and cloud. In *Proceedings of the sixth conference on Computer systems*, pages 301–314. ACM, 2011.

[8] Ericsson ConsumerLab. Smartphone usage experience. 2013.

[9] Robert Harry Crites. *Large-scale dynamic optimization using teams of reinforcement learning agents*. PhD thesis, Citeseer, 1996.

[10] Prashant Doshi and Dennis Perez. Generalized point based value iteration for interactive pomdps. In *AAAI*, pages 63–68, 2008.

[11] Jason Flinn and Mahadev Satyanarayanan. Powerscope: A tool for profiling the energy usage of mobile applications. In *Mobile Computing Systems and Applications, 1999. Proceedings. WMCSA'99. Second IEEE Workshop on*, pages 2–10. IEEE, 1999.

[12] Marek Grześ, Pascal Poupart, and Jesse Hoey. Controller compilation and compression for resource constrained applications. In *Algorithmic Decision Theory*, pages 193–207. Springer, 2013.

[13] Marek Grześ, Pascal Poupart, and Jesse Hoey. Isomorph-free branch and bound search for finite state controllers. In *Proceedings of the Twenty-Third international joint conference on Artificial Intelligence*, pages 2282–2290. AAAI Press, 2013.

[14] Marek Grześ, Pascal Poupart, Xiao Yang, and Jesse Hoey. Energy efficient execution of pomdp policies. Submitted to SMC for review, 2014.

[15] Milos Hauskrecht and Peter Szolovits. *Planning and control in stochastic domains with imperfect information*. PhD thesis, Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science, 1997.

[16] Jesse Hoey, Robert St-Aubin, Alan J Hu, and Craig Boutilier. Spudd: Stochastic planning using decision diagrams, 1999.

[17] Jesse Hoey, Axel von Bertoldi, Pascal Poupart, and Alex Mihailidis. Assisting persons with dementia during handwashing using a partially observable markov decision process, 2007.

[18] Jesse Hoey, Xiao Yang, Eduardo Quintana, and Jesús Favela. Lacasa: location and context-aware safety assistant. In *Pervasive Computing Technologies for Healthcare (PervasiveHealth), 2012 6th International Conference on*, pages 171–174. IEEE, 2012.

[19] Dijiang Huang et al. Mobile cloud computing. *IEEE COMSOC Multimedia Communications Technical Committee (MMTC) E-Letter*, 6(10):27–31, 2011.

[20] Leslie Pack Kaelbling, Michael L Littman, and Anthony R Cassandra. Planning and acting in partially observable stochastic domains. *Artificial intelligence*, 101(1):99–134, 1998.

[21] Rodney M LaFollette. Design fundamentals of high-power-density, pulsed-discharge, lead-acid batteries. Technical report, Brigham Young Univ., Provo, UT (USA), 1988.

[22] George E Monahan. State of the arta survey of partially observable markov decision processes: Theory, models, and algorithms. *Management Science*, 28(1):1–16, 1982.

[23] Dushyanth Narayanan, Jason Flinn, and Mahadev Satyanarayanan. Using history to improve mobile application adaptation. In *Mobile Computing Systems and Applications, 2000 Third IEEE Workshop on.*, pages 31–40. IEEE, 2000.

[24] Robert F Nelson, Richard Rinehart, and Sean Varley. Ultrafast pulse discharge and recharge capabilities of thin-metal film battery technology. In *Pulsed Power Conference, 1997. Digest of Technical Papers. 1997 11th IEEE International*, volume 1, pages 636–641. IEEE, 1997.

[25] Cat Phone. New research reveals mobile users want phones to have a longer than average battery life, 2013.

[26] Martha E Pollack, Laura Brown, Dirk Colbry, Cheryl Orosz, Bart Peintner, Sailesh Ramakrishnan, Sandra Engberg, Judith T Matthews, Jacqueline Dunbar-Jacob, Colleen E McCarthy, et al. Pearl: A mobile robotic assistant for the elderly. In *AAAI workshop on automation as eldercare*, volume 2002, pages 85–91, 2002.

[27] Pascal Poupart. *Exploiting structure to efficiently solve large scale partially observable Markov decision processes*. PhD thesis, Citeseer, 2005.

[28] Pascal Poupart. An introduction to fully and partially observable markov decision processes. *Decision Theory Models for Applications in Artificial Intelligence: Concepts and Solutions, IGI Global*, pages 33–62, 2011.

[29] Balakrishna J Prabhu, A Chockalingam, and Vinod Sharma. Performance analysis of battery power management schemes in wireless mobile devices. In *Wireless Communications and Networking Conference, 2002. WCNC2002. 2002 IEEE*, volume 2, pages 825–831. IEEE, 2002.

[30] Martin L Puterman. *Markov decision processes: discrete stochastic dynamic programming*, volume 414. John Wiley & Sons, 2009.

[31] Anand Raghunathan and Kanishka Lahiri. Battery life estimation of mobile embedded systems. *VLSI Design*, 2001.

[32] Venkat Rao, Gaurav Singhal, Anshul Kumar, and Nicolas Navet. Battery model for embedded systems. In *VLSI Design, 2005. 18th International Conference on*, pages 105–110. IEEE, 2005.

[33] John M Rulnick and Nicholas Bambos. Mobile power management for maximum battery life in wireless communication networks. In *INFOCOM'96. Fifteenth Annual Joint Conference of the IEEE Computer Societies. Networking the Next Generation. Proceedings IEEE*, volume 2, pages 443–450. IEEE, 1996.

[34] Jeff Sharkey. Coding for life–battery life, that is. In *Google IO Developer Conference*, 2009.

[35] Richard D Smallwood and Edward J Sondik. The optimal control of partially observable markov processes over a finite horizon. *Operations Research*, 21(5):1071–1088, 1973.

[36] Matthijs TJ Spaan, Nelson Gonçalves, and João Sequeira. Multiagent coordination by auctioning pomdp tasks. In *Proceedings of the 4th Multi-agent Sequential Decision Making in Uncertain Domains Workshop*, pages 44–55, 2009.

[37] Chang Tan, Qi Liu, Enhong Chen, and Hui Xiong. Prediction for mobile application usage patterns.

[38] Intelligent Assistive Technology and Systems Lab (IATSL). The coach (cognitive orthosis for assisting with activites in the home), 2013.

[39] Chelsea C White III. Optimal diagnostic questionnaires which allow less than truthful responses. *Information and Control*, 32(1):61–74, 1976.

[40] Jason D Williams and Steve Young. Partially observable markov decision processes for spoken dialog systems. *Computer Speech & Language*, 21(2):393–422, 2007.

[41] Xiao Yang. Intelligently dividing tasks between mobile devices and the cloud. University of Waterloo, 2012. Course project of CS854, Cloud Computing.

[42] Wanghong Yuan and Klara Nahrstedt. Energy-efficient soft real-time cpu scheduling for mobile multimedia systems. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 149–163. ACM, 2003.